***COMPLEXITY ANALYSIS***

**1)** The `increaseKey` function has a time complexity of O(log n):

|  |  |
| --- | --- |
|  |  |
| **1 function increaseKey(int i, TaskReminder key) {**  **2** if (key.getImportanceLevel() < heap[i].getImportanceLevel()) {  **3** return;  **4** }  **5** heap[i] = key;  **6** while (i > 0 && heap[i].getImportanceLevel() > heap[parent(i)].getImportanceLevel()){  **7** swap(i, parent(i));  **8** i = parent(i);  **9** }  **10 }** | **1** O(1)  **2** O(1)  **3** O(1)  **4**  **5** O(1)  **6** O(log n)  **7** O(1)  **8** O(1)  **9**  **10** |

Most lines of code have a time complexity of O(1), except for line 5, which involves a while loop with a worst-case time complexity of O(log n), where "n" is the number of elements in the heap. The overall complexity of the `increaseKey` function is O(log n) in the worst case due to the while loop.

**2)** The `delete` function has a time complexity of O(n):

|  |  |
| --- | --- |
|  |  |
| **1 function delete(K key) {**  **2** int index = hash(key);  **3** Node<K, V> currentNode = table[index];  **4** Node<K, V> previousNode = null;  **5** while (currentNode != null) {  **6** if (currentNode.getKey().equals(key)) {  **7** if (previousNode != null) {  **8** previousNode.setNext(currentNode.getNext());  **9** } else {  **10** table[index] = currentNode.getNext();  **11** }  **12** currentNode.setNext(null);  **13** return;  **14**  }  **15** previousNode = currentNode;  **16** currentNode = currentNode.getNext();  **17** }  **18 }** | **1** O(1)  **2** O(1)  **3** O(1)  **4** O(1)  **5** O(n)  **6** O(1)  **7** O(1)  **8** O(1)  **9** O(1)  **10** O(1)  **11**  **12** O(1)  **13** O(1)  **14**  **15** O(1)  **16** O(1)  **17**  **18** |

The while loop in line 5, which has a time complexity of O(n) in the worst case, where "n" is the length of the linked list. The overall time complexity of the `delete` function is O(n) in the worst case due to the while loop. This means that the execution time increases linearly with the size of the linked list, making the function efficient for removing elements from a linked list.

**Spatial complexity analysis**

**1) public Node<K, V> search(K key){**

int index = hash(key);

if (table[index] != null){

Node<K, V> currentNode = table[index];

while (currentNode != null) {

if (currentNode.getKey().equals(key)) {

return currentNode;

}

currentNode = currentNode.getNext();

}

}

return null;

**}**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Variable** | **Size of 1 atomic** | **Quantity of atomic** |
| Input | key | 64 bits | 1 |
| Auxiliary | Index | 32 bits | 1 |
| currentNode | 32 bits | 1 |
| Output | resultNode | 32 bits | 1 |

**Total spatial complexity =** Input + Auxiliary + Output = 4 = θ(1)

**Auxiliary spatial complexity =** 2 = θ(1)

**Auxiliary + Output spatial complexity =** 1 + 1 = θ(1)

**2) public void increaseKey(int i, TaskReminder key) {**

if (key.getImportanceLevel() < heap[i].getImportanceLevel()) { 🡪 O(1)

return; 🡪 O(1)

}

heap[i] = key; 🡪 O(1)

while (i > 0 && heap[i].getImportanceLevel() > heap[parent(i)].getImportanceLevel()) { 🡪 O(log n)

swap(i, parent(i)); 🡪 O(1)

i = parent(i); 🡪 O(1)

}

**}**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Variable** | **Size of 1 atomic** | **Quantity of atomic** |
| **Input** | **i** | **32 bits** | **1** |
| **key** | **64 bits** | **1** |
| **Auxiliary** | **i** | **32 bits** | **1** |

**Total spatial complexity = Input + Auxiliary + Output = 1 + 1 + 1+ 1 = θ(1)**

**Auxiliary spatial complexity = 1 = θ( 1)**

**Auxiliary + Output spatial complexity = 1 + 0= θ(1)**